Chapter IV

UniFrame:
A Unified Framework for Developing Service-Oriented, Component-Based Distributed Software Systems

Rajeev R. Raje, Indiana University Purdue University, USA

Andrew M. Olson, Indiana University Purdue University, USA

Barrett R. Bryant, University of Alabama at Birmingham, USA

Carol C. Burt, University of Alabama at Birmingham, USA

Makhail Auguston, Naval Postgraduate School, USA

Abstract

This chapter introduces the UniFrame approach to creating high quality computing systems from heterogeneous components distributed over a network. It describes how this approach employs a unifying framework for specifying such systems to unite the concepts of service-oriented architectures, a component-based software engineering methodology and a mechanism for automatically finding components on a network in order to assemble a specified system. UniFrame employs a formal specification language to define the components and serve as a basis for generating glue/wrapper code that connects heterogeneous components. It also provides a high level language for the system developer to use for inserting code in a created system to validate it

1. REPORT DATE **2005**	2. REPORT TYPE	3. DATES COVERED **00-00-2005 to 00-00-2005**
4. TITLE AND SUBTITLE **Uniframe: A Unified Framework for Developing Service-Oriented, Component-Based Distributed Software Systems**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Naval Postgraduate School,Department of Computer Science,Monterey,CA,93943**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Service-Oriented Software System Engineering: Challenges and Practices (Chapter IV, pp. 68-87). Hershey,PA: Idea Group Publishing

14. ABSTRACT
see report

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**20**	

empirically and estimate the quality of service it supports. The chapter demonstrates how a comprehensive approach, which involves the practicing community as well as technical experts, can lead to solutions of many of the difficulties inherent in constructing distributed computing systems.

Introduction

The architecture of a computing system family can be represented by a business model comprising a set of standard, platform independent models residing in a service layer, each of which is related to a platform specific model that corresponds to one or more specific realizations of the service. A system is realized by assembling the realizations according to the specified architecture. This Service-Oriented Architecture offers many advantages, such as flexibility, in constructing and modifying a computing system. Because business requirements can change rapidly, both the services making up a business model and their platform specific realizations may need to change rapidly in response. With an agile mechanism to trace out an appropriate architecture, the development engineer can react quickly by building a modified realization of the system. Nevertheless, there are many practical issues that make effecting this process difficult. For example, an environment in which this approach has greatest appeal is typically distributed and heterogeneous. This makes the mapping of a system's platform independent model to a platform specific model (Object Management Group, 2002) quite complex and subject to variation.

This chapter describes the basic principles of the UniFrame Project, which defines a process, based on Service-Oriented Architecture, for rapidly constructing a distributed computing system that confronts many of these inherent difficulties. UniFrame's basic objective is to create a unified framework to facilitate the interoperation of heterogeneous distributed components as well as the construction of high quality computing systems based on them. UniFrame combines the principles of distributed, component-based computing, Model-Driven Architecture, service and quality of service guarantees, and generative techniques.

Though better than handcrafting distributed computing systems, developing them by composing existing components still poses many challenges. A comprehensive treatment of these and the corresponding solutions that UniFrame proposes exceeds the scope of this chapter, so it sketches the features of UniFrame that are most related to the book's service-oriented engineering theme along with references to further reading.

Background

Despite the achievements in software engineering, development of large-scale, decentralized systems still poses major issues. Recent experience has demonstrated that the

principles of distributed, component-based engineering are effective in dealing with them. Weck (1997), Lumpe, Schneider, Nierstrasz, and Achermann (1997), and the works of Batory et al., for example, Batory and Geraci (1997), concern the composition of components. The approach of Griss (2001) to developing software product lines is similar to UniFrame's, except that UniFrame avoids descending to code-fragment-sized components. Brown (1999) surveys component-based system development, whereas Heineman and Councill (2001) and Szyperski, Gruntz, and Murer (2002) provide extensive discussions of different aspects.

Heineman and Councill (2001) provide a general definition of a component model. Many different models for distributed, component-based computing have been proposed and implemented. Among these, J2EE™ (Java 2 Enterprise Edition) and its associated distributed computing model (Java-RMI), CORBA® (Common Object Request Broker Architecture), and .NET® have achieved the greatest acceptance. Typically, each prevalent model assumes the presence of homogeneous environments; that is, components created using a particular model assume that any other components present adhere to the same model. For example, the white paper on Java Remote Method Invocation (2003) describes RMI as an extension of Java's basic model to achieve distributed computation, assuming, thus, an environment consisting of components developed using Java and communicating with each other using method calls. Schmidt (2003) provides an overview of CORBA, which indicates that CORBA does provide a limited independence from the components' development language and deployment platform by specifying components with an interface definition language. This permits implementation in any languages for which mappings with the interface definition language exist. Again, an implicit assumption is that, typically, a CORBA component will communicate with another CORBA component. Microsoft's .NET is intended as a programming model for building XML™-based Web services and associated applications. It provides language independence with an interface language and a common language runtime (Microsoft .NET Framework, 2003). The implicit assumption of homogeneity still holds.

UniFrame

Current approaches for tackling heterogeneity are *ad hoc* in nature, requiring handcrafted software bridges so have many drawbacks. It is difficult to make components of different models interoperate, and handcrafting is known to be error prone. Moreover, dependence on a single model meshes poorly with the grand notion of a component (or services) bazaar over a distributed infrastructure, as the success of such a bazaar requires local autonomy for deciding various policies, including the choice of the underlying model. Thus, there is a need for a framework, such as UniFrame, that will support seamless interoperation of heterogeneous, distributed components. UniFrame consists of:

*   the creation of a standards-based meta-model for components and associated hierarchical setup for indicating the contracts and constraints of the components;

- an automatic generation of glue and wrappers for achieving interoperability;

- guidelines for specifying and verifying the quality of individual components;

- a mechanism for automatically discovering appropriate components on a network;

- a methodology for developing distributed, component-based systems with service-oriented architectures; and

- mechanisms for evaluating the quality of the resulting component assemblages.

UniFrame creates more general distributed systems than the point-to-point interactions of current Web services and also emphasizes determining the Quality of Service (QoS) during system assembly. For pragmatic reasons, UniFrame provides an iterative, incremental process for assembling a distributed computing system (DCS) from services available on the network that permit selecting among alternative components during system construction. In order to increase the assurance of a DCS, UniFrame employs automation, to the extent feasible, in the processes of locating and assembling components, and of component and system integration testing. The ICSE 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction (Crnkovic, Schmidt, Stafford & Wallnau, 2003) focused on automated composition theories in constructing a DCS. Although automation is a goal of UniFrame, it presently focuses on the more practical, implementation aspects.

Unified Meta-Component Model (UMM)

Because future service-oriented systems will consist of independently developed components adhering to various models, a meta-model that abstracts the features of different models, enhances them and incorporates innovative concepts, is necessary in order to facilitate their creation. Raje (2000) and Raje, Auguston, Bryant, Olson, and Burt (2001) describe a central concept of UniFrame, the Unified Meta-component Model, that does this. It consists of three parts: (a) components, (b) service and its guarantees, and (c) infrastructure. These are not novel separately, but their structure, integration, and interactions form the UMM's distinguishing features. Components in the UMM have public interfaces and private implementations, which may be heterogeneous. Each interface comprises multiple levels. In addition to emphasizing a component's functional responsibilities (or the services it offers), the UMM requires component developers to advertise and guarantee a QoS rating for each component. The UMM's infrastructure supplies the environment necessary for developing, deploying, publishing, locating, assembling, and validating individual components and systems of components. The following subsections expand upon these concepts.

Component

The UMM defines a component as a sextuple consisting of the attributes (inherent, functional, nonfunctional, cooperative, auxiliary, deployment). This view of a component conforms to the definition of Szyperski, Gruntz, and Murer (2002). The inherent attributes contain the bookkeeping information about a component, such as the author, the version, and its validity period. The functional attributes of a component contain its interface, along with the necessary pre- and post-conditions, and component model of any associated implementation. They also indicate related details, such as algorithms used, underlying design patterns and technology, and known usages. The nonfunctional attributes represent the QoS parameters supported by the component, along with their values that the component developer guarantees in a specific deployment environment. These attributes may also indicate the effects of the deployment environment and usage patterns on the QoS values. The cooperative attributes describe how components actively collaborate, exchanging services. The auxiliary attributes exhibit other characteristics, such as mobility, various security features, and fault tolerance that the components may possess. A component needs deployment rules, specified in the deployment attributes so that it can be configured, initialized, and made available on a network.

Service

As described by Raje (2000), this part of the UMM consists of the computational tasks and guarantees that a component performs. To realize a DCS from a set of independently created components, the system integrator needs to reason from the service assurance of each component to obtain the assurance of the integrated DCS. Hence, a component must provide a predetermined level of assurance of both its functional and nonfunctional features. Various techniques, such as formal verification, have been proposed for reasoning about the functional assurance of a DCS. Therefore, the UMM assumes the use of an appropriate mechanism for functional assurance. The UniFrame research focuses on assuring the nonfunctional features of components and the integrated system because many existing application domains (multimedia, critical systems, and so forth) depend not only on correct functionality but also on how well it is achieved. UniFrame provides a mechanism for the component provider to specify the QoS parameters that are applicable to a provided component and determine the ranges that the component can guarantee.

Table 1 shows the UMM type specification of a component, *Validation Server*, for validating user accesses within the application domain of document management. In the advertised description of a corresponding implementation, the component provider would supply the actual values for various fields (such as N/A in Table 1). For example, the specification of a component that implements *Validation Server* would contain details, such as the URL where the component is deployed (id), the guaranteed values for the *throughput* and *end-to-end delay*, and the required deployment environment. The

Table 1. UMM type specification of a component

Abstract Component Type: *ValidationServer*

1. Component Name: *ValidationServer*
2. Domain Name: Document Management
3. System Name: DocumentManager
4. Informal Description: Provide the user validation service.
5. Computational Attributes:
    5.1 Inherent Attributes:
        5.1.1 id: N/A
        5.1.2 Version: version 1.0
        5.1.3 Author: N/A
        5.1.4 Date: N/A
        5.1.5 Validity: N/A
        5.1.6 Atomicity: Yes
        5.1.7 Registration: N/A
        5.1.8 Model: N/A
    5.2 Functional Attributes:
        5.2.1 Function description: Act as validation server for users in the system.
        5.2.2 Algorithm: N/A
        5.2.3 Complexity: N/A
        5.2.4 Syntactic Contract
        5.2.4.1 Provided Interface: *IValidation*
        5.2.4.2 Required Interface: NONE
        5.2.5 Technology: N/A
        5.2.6 Expected Resources: N/A
        5.2.7 Design Patterns: NONE
        5.2.8 Known Usage: Validation of user access
        5.2.9 Alias: NONE
6. Cooperation Attributes:
    6.1 Preprocessing Collaborators: *Users'Terminal*
    6.2 Postprocessing Collaborators: NONE
7. Auxiliary Attributes:
    7.1 Mobility: No
    7.2 Security: *L0*
    7.3 Fault tolerance: *L0*
8. Quality of Service Attributes
    8.1 QoS Metrics: *throughput*, *end-to-end delay*
    8.2 QoS Level: N/A
    8.3 Cost: N/A
    8.4 Quality Level: N/A
    8.5 Effect of Environment: N/A
    8.6 Effect of Usage Pattern: N/A
9. Deployment Attributes: N/A

specification associated with each implemented component is published when it is deployed on the network. The UMM specification of a component enhances the concept of a multilevel contract for components proposed by Beugnard, Jezequel, Plouzeau, and Watkins (1999) *because it includes other details, such as bookkeeping, collaborative, algorithmic and technological information, and possible levels of service with associated costs and effects of different environmental factors on the QoS parameters.*

Infrastructure

UniFrame assumes the presence of a publicly accepted knowledgebase that contains information, such as the component types needed for a specific application domain, the interconnections and constraints that make up the design specification of each component system in a domain, and rules for QoS calculations. Experts, such as standards organizations' task forces, create the UMM specifications for the components of each application domain of the knowledgebase. The UMM specifications of the component types are publicly distributed so that component developers can supply implementations that adhere to them.

UniFrame's Infrastructure consists of the System Generation Process, Resource Discovery Service (URDS), and Glue and Wrapper Generator. The first employs the knowledgebase to carry out the steps in creating a component system. It invokes the URDS to locate the components in the network the system requires and validates the product using an iterative process. The URDS provides mechanisms for components to publish their UMM specifications and for hosting the services on distributed machines, receives appropriate queries for locating the deployed services, and performs the selection of necessary components based upon specified criteria. It invokes the Glue and Wrapper Generator, which accommodates the heterogeneity across components, incorporates the mechanisms necessary to measure the QoS, and configures the selected services. Subsequent sections will provide more details about these.

Service-Oriented Architecture

In order to provide flexible, efficient support to the process of creating a DCS, UniFrame organizes its knowledgebase according to the concepts of Model-Driven Architecture proposed by the Object Management Group (2002) and Business Line Architecture proposed by Enterprise Architecture SIG (2003a). Its UMM provides an underlying framework for this organization. The domain elements in the top tier of the architecture correspond to different business contexts, or lines. A context consists of a class of related business practice domains (such as, retail grocery, retail hardware, construction supply, wholesaler), which are located in the next tier down. Conceptually, elements on one level can share an element on another (health care and construction can share inventory), which differs in how it performs similar operations in different contexts (that is, the element comprises a set of variants). The various, hierarchically organized elements that contribute detail to the definition of a business context constitute its Business Reference Model, discussed in Succeeding with Component-Based Architecture by Enterprise Architecture SIG (2003b). This takes the form of a tree, whose root represents the context in the architecture under consideration. Business domain experts perform requirements analysis and model the business contexts for which it is desired to construct DCSs. The Business Reference Models they derive and place in the knowledgebase define the space of problems UniFrame can solve.

For each Business Reference Model, software engineers construct design models in various ways to implement DCSs that satisfy its requirements. A design model is expressed, frequently in Unified Modeling Language (UML®) (Rumbaugh, Jacobson & Booch, 1999), in terms of tiered layers of components, each component offering a defined set of services. Several Business Reference Models can share components. A component in one tier can be composed (or use of the services) of components on a lower tier. Thus, a component has two definition forms in the knowledgebase:

- a specification of its abstract properties as a type, as in Table 1, or

- a design specification, following UMM standards, that directly references the components and refined design specifications of which it uses.

The former is called an *abstract component*, which the UniFrame System Generation Process considers to be available with no construction necessary. The second form is called a *compound component*. The process will attempt to construct it from its design. A design specification that defines a realization of a Business Reference Model forms a Service Reference Model for it. It provides a vehicle for realizing the Model-Driven Architecture's mapping from a platform-independent model to a platform-specific model. The Service Reference Models also form part of UniFrame's knowledgebase.

In order to construct DCS solutions for a significant space of problems, the knowledgebase must contain matching (Business Reference Model, Service Reference Model) pairs for each problem variation anticipated. These can be organized efficiently by structuring related Business Reference Models in feature models according to the optional features that they exhibit and related Service Reference Models according to variation point stereotypes that show which design variants are available. The experts create a domain-specific language based on the distinguishing features and variation points in the models. Then, users of the System Generation Process employ the language to specify their requirements. The following example illustrates the knowledgebase's organization.

Case Study

Suppose domain experts want to create a knowledgebase that includes the business context consisting of users who manage documents. The users' contact with the supporting system is via the use case *Manage Documents*, which includes *Validate User*. The use cases *Create Document, Delete Document*, *List Documents*, *Store Document,* and *Get Document* all extend *Manage Documents*. The last in this list includes *Lock Document*, whereas the others include *Unlock Document*. From the requirements these express, the domain experts identify three subsystems comprising the system: one for user validation, one for managing the documents themselves, and one for user interaction. The experts write a domain model for this system containing these three subsystems.

Suppose the experts decide the users may want to choose between two types of document manager systems: a standard document manager and a deluxe one that provides extended persistence support. They represent these options in a simplified feature diagram for the document manager, as shown in Figure 1. Clear small circles indicate optional features, whereas an arc indicates an exclusive OR choice. In more general feature diagrams (Griss, 2001), options of a node can be chosen as any combination of elements of a subset of the node's children. A feature diagram carries no information about how its alternatives might be associated with elements in the domain model of their parent node. It is an efficient mechanism for representing alternatives; the domain models are essential for representing the associations among elements in the models and the constraints on them. The domain model for the standard document manager consists of only one domain element, *Document Server*. The domain model for the deluxe document manager consists of two domain elements, *Deluxe Document Server* and its associated *Document Database* for persistence. Because there are just two alternatives in the feature diagram, there are just two Business Reference Models in this example. More generally, there will be as many as there are combinations permitted by the various feature diagrams present in the knowledgebase.

Software engineers experienced in the domain of the business context (document management here) develop design models for these two Business Reference Models. They create a service-oriented architecture of abstract components so that domain models map to component-based design models. Figure 2 shows the Service Reference Model, *Standard Document System*, for the Business Reference Model of the *Standard Document Manager* for this example. The Service Reference Model, *Deluxe Document System*, for the *Deluxe Document Manager* is identical, with the addition of a *Database* component associated with the *Document Server*, where the cardinality allows an arbitrary, positive number of *Database* units to be present. The Service Reference Models include the details defining the associations among the components. These might be views consisting of UML collaboration diagrams. This information is used to determine the entries in the UMM abstract component specifications and the interrelations of the components' interfaces. The specification for the abstract component, *Validation Server*, appeared in Table 1.

Suppose that the software engineers decide that two implementations of the standard document manager are possible, one in which the components adhere to .NET and the

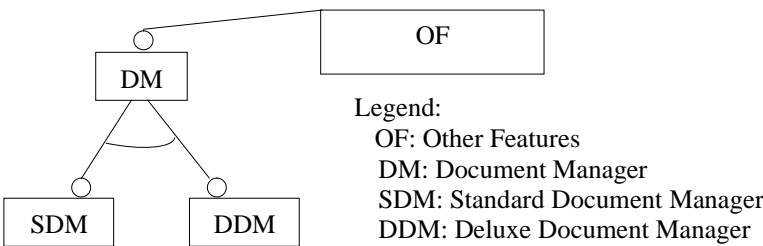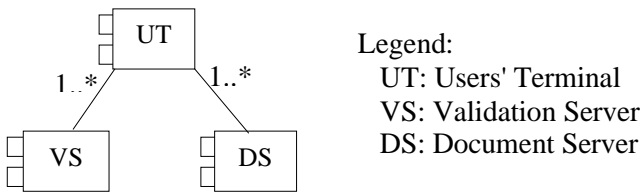*Figure 1. Feature diagram for the document management system*

Legend:
  OF: Other Features
  DM: Document Manager
  SDM: Standard Document Manager
  DDM: Deluxe Document Manager

Figure 2. Service reference model for the standard document system

Legend:
UT: Users' Terminal
VS: Validation Server
DS: Document Server

other to CORBA. They indicate this choice by a design model, labeled *Standard Document System*, augmented by variation point information that specifies the choice of one of these two technologies for the associations in Figure 2, such as in OCL (Warmer & Kleppe, 2003), as shown:

context Standard Document System
inv: technology = '.NET' or technology = 'CORBA'

Because the system consists of more than two components, the engineers have other combinations possible. For example, the *Users' Terminal/Validation Server* association may be in .NET technology, and the *Document Server* may be in CORBA technology, implying the need for an appropriate bridge.

UniFrame System Generation Process

The essential steps in UniFrame's process of constructing a DCS to solve a problem appear in Table 2. Once the UniFrame knowledgebase is available, a system developer can pose a statement of requirements for a DCS that solves a problem within its application domain. This analysis task forms step (1) in Table 2. For the case study in the previous section, the statement of requirements might be:

Create a Document Management System having a Standard Document Manager.

In step (2), the term *Document Management System* of the example requirements statement identifies the business context, so the stated problem lies within the domain the knowledgebase represents. The corresponding system model shows there are two alternatives for the *Document Manager,* which the feature model displays in Figure 1. The qualifying requirement, *Standard,* resolves this ambiguity, which completes step (2). The resulting Business Reference Model maps directly in the knowledgebase to the two alternative platform-specific Service Reference Models for the entire system shown in

Table 2. Steps in the UniFrame System Generation Process

Steps	Activities
1	State the requirements the DCS must satisfy in the knowledgebase's terminology.
2	Identify a Business Reference Model that represents these.
3	Identify each Service Reference Model specifying a system of abstract components that satisfies the Business Reference Model.
4	Obtain concrete implementations of the abstract components.
5	Assemble the concrete components into a DCS according to each Service Reference Model, so that it meets the specified requirements.
6	Test the DCS against the requirements and exit if satisfactory; otherwise, return to step (1) to modify the requirements.

Figure 2, in which the components are either all .NET or all CORBA. This completes step (3).

Continuing to step (4), the System Generation Process collects the UMM type specifications of all the abstract components involved in each of the two Service Reference Models and sends them in a query to the UniFrame Resource Discovery Service. This searches the network for implemented components whose UMM descriptions satisfy the type specifications.

Step (5) employs the design information in a Service Reference Model to construct a DCS with the components found. If the appropriate implementations are available on the network, the request for a *Standard Document Manager* in the example will yield two DCSs, one with .NET technology and one with CORBA technology. If no .NET implementation of a *Validation Server* is found, then only the CORBA DCS will be constructed.

Typically, a developer understands the requirements poorly at the initiation of the System Generation Process. Therefore, it is imperative to evaluate empirically the consistency of the characteristics of a generated DCS with the perceived requirements and make modifications as necessary. This motivates having step (6) in Table 2. Such iterative development provides a mechanism for the developer to validate the outcome of the process and determine empirically the ranges within which its QoS attributes vary. This helps to assure a higher quality product. The process allows two levels of testing. The simplest is black box (or acceptance) testing of the DCS based on only the stated requirements. The developer supplies a test harness and plan for this. The other is white box (or integration) testing, again based on the developer's test plan. In this case, the design of the DCS serves as a guide for inserting instrumentation code between the components in the DCS. At runtime, this code reports the behavior of the DCS, giving the developer a view into its internal operation. The section on the measurement of QoS discusses a mechanism for inserting this instrumentation easily.

In case there are several Business or Service Reference Models in the knowledgebase that satisfy the developer's requirements if step (2) or (3) of the process provides feedback, allowing the developer to introduce requirements incrementally so as to reduce

these alternatives, then the process becomes an efficient way to construct the needed type of DCS. Thus, the System Generation Process supports the iterative, incremental development paradigm that modern software engineering practices have found productive.

UniFrame Resource Discovery Service (URDS)

Once components and their UMM descriptions have been deployed on the network, they are ready for discovery in the UniFrame System Generation Process. The URDS executes this process. Siram et al. (2002) discusses its architecture, shown in Figure 3.

The URDS architecture comprises: HeadHunters (HHs), Internet Component Broker (ICB), Meta-repositories (MRs), and components.

Components are implemented according to some component model, as described earlier, and registered with the model's binding service. For example, the Java-RMI components are registered with the Naming service provided by the Java-RMI framework. An

Figure 3. UniFrame Resource Discovery System (URDS)
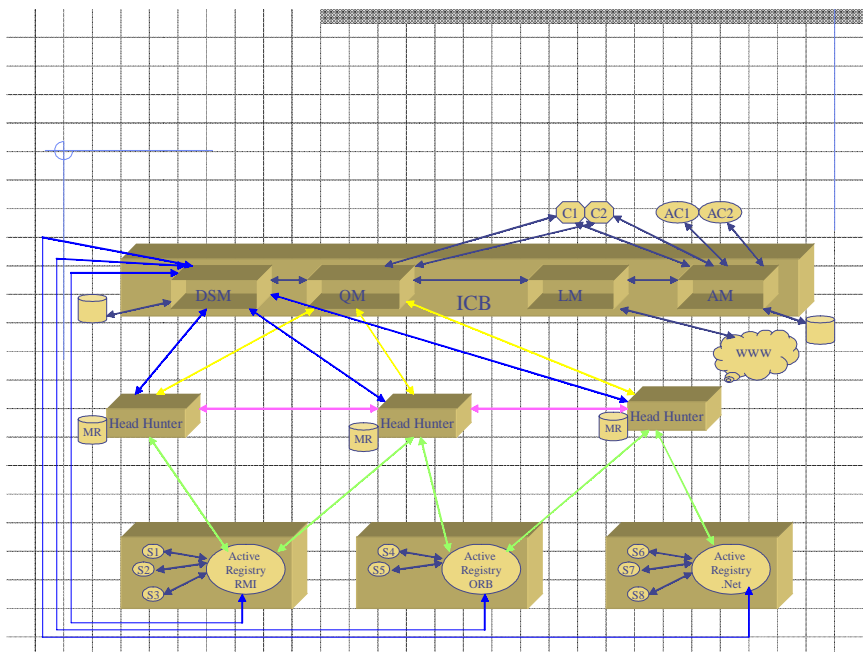
advantage of this is that it does not burden the component providers as to deploy their implementations, they must register them anyway. The HHs have the sole responsibility of performing matchmaking operations between registered components and requested specifications. Each HH has an MR, which serves as a local store. An HH is constantly discovering newly implemented components and storing their UMM specifications in its MR. Anytime an HH receives a query for a component type, it first searches its MR. If it finds a match, it returns the corresponding component as a result. If not, it propagates the query to other HHs in the system.

The ICB is analogous to the object request broker (ORB) in other architectures. Unlike the ORB, which only allows interoperation between components having heterogeneous implementations, the Internet component broker allows interoperation between components with different component models. As Figure 3 shows, the Internet component broker consists of domain security manager (DSM), query manager (QM), link manager (LM), and adapter manager (AM). The DSM is responsible for enforcing a security structure on the URDS. It authenticates the HHs and allows them to communicate with different binding mechanisms (registries). The QM interfaces with the System Generation Process. It receives a query consisting of a collection of UMM component types, passes it to the HHs, and returns the results. The LM allows a federation of URDSs to be created in order to increase the component search space. The AM locates adapter components, such as bridges that allow interoperation of different component models, and passes them to the Glue and Wrapper Generator.

A prototype of URDS has been implemented using the Java-RMI and .NET technologies. Many experiments have been performed to measure its performance (Siram et al., 2002). These demonstrate that URDS scales upward, but the details extend beyond this chapter's scope.

Industry and academia have proposed and implemented many distributed resource discovery and directory services. Examples that Siram et al. (2002) describe include WAIS, Archie, Gopher, UDDI, CORBA Trader, LDAP, Jini, SLP, Ninf, and NetSolve. Each has its own characteristics and exhibits some similarity with URDS. The distinguishing features of URDS are its treatment of heterogeneity and its purpose to support creating heterogeneous integrated systems, not just to discover services.

UniFrame Quality of Service Framework (UQoS)

Components offer services and indicate and guarantee the quality of their services. Therefore, it is necessary to facilitate the publication, selection, measurement, and validation of component and DCS QoS values. The UniFrame Quality of Service Framework, described by Brahnmath (2002); Sun (2003); and Raje, Bryant, Olson, Auguston, and Burt (2002), provides necessary guidelines for the component developers and system integrators using UniFrame. The UQoS consists of three parts: QoS catalog,

composition/decomposition models for QoS parameters, and specification and measurement of QoS. The reader is referred to the references above for the first two because the details are extensive.

To prepare the UMM description of a component to be publicized, the component developer must measure empirically the QoS parameters in the corresponding UMM type specification. The QoS catalog provides model definitions and formulas to assist in this. Some parameters are static in nature (like reliability), while some are dynamic (like end-to-end delay). If the parameter is static and characterizes a system of components, then its value can be determined from the components' parameter values. Otherwise, its value must be determined empirically.

Evaluation of QoS Parameters

UniFrame uses the principles of event grammars for measuring parameters empirically. Event grammar, as described by Auguston (1995), forms the basis for system behavior models. An event represents any detectable action during execution, such as a statement execution, expression evaluation, procedure call, and receiving a message. It has a beginning, end, and duration (a time interval corresponding to the action of interest). Actions (or events) evolve in time, and system behavior represents the temporal relationship among actions. This implies a partial ordering relation for events, as Lamport (1978) discussed.

System execution can be modeled as a set of events (event trace) with two basic relations: partial ordering and inclusion. The event trace actually is a model of the system's temporal behavior. In order to specify meaningful system behavior properties, events must be enriched with attributes. An event may have a type and other attributes, such as duration, source code related to the event, associated state (that is, variable values at the event's beginning and end), and function name and returned value for function call events.

A special programming language, FORMAN, for computations over event traces greatly facilitates measuring parameters empirically. As described by Fritzson, Auguston, and Shahmehri (1994) and Auguston (1995), it is based on the notions of the functional paradigm, event patterns, and aggregate operations over events.

The execution model of a component (or a system of integrated components) is defined by an event grammar, which is a set of axioms that describes possible patterns of basic relations between events of different types in a program execution trace. It is not intended to be used for parsing actual event traces. If an event is compound, the grammar describes how it splits into other event sequences or sets. For example, the event *execute-assignment-statement* contains a sequence of events *evaluate-right-hand-part* and *execute-destination*.

The rule *A :: (B C)* establishes that, if an event *a* of the type *A* occurs in the trace of a program, it is necessary that events *b* and *c* of types *B* and *C*, also exist, such that the relations *b* IN *a*, *c* IN *a*, *b* PRECEDES *c* hold. For example, the event grammar describing

the semantics of an imperative programming language may contain the following rule (the names, such as *execute-program* and *ex-stmt* in the grammar denote event types):

$$execute\text{-}program \ :: (\ ex\text{-}stmt \ * \ )$$

This means that each event of the type *execute-program* contains an ordered (w.r.t. relation PRECEDES) sequence of zero or more events of the type *ex-stmt*. For the function call event, the event grammar may provide the following rule:

$$func\_call:: (param \ * \ ) \ (\ ex\text{-}stmt \ * \ )$$

This event may contain zero or more parameter evaluation events followed by statement executions.

Example of Evaluating Turn-Around Time

If the event type *component_call* corresponds to the whole component call event and *request* denotes the event for a single request (the time interval from the request's beginning to its completion), then the following FORMAN formula specifies the measurement of the turn-around time:

FOREACH a: session FROM execute_program
SAY ('Turn-around Time for a session is '
     SUM[b: request FROM a APPLY b.duration]
                / CARD[request FROM a])

Similar rules can be specified for any other dynamic QoS parameters or related computations. Thus, the principles of event traces provide a mechanism to validate empirically the QoS values for a component and for an integrated system of components.

Interoperability Using the
Glue and Wrapper Generator

For interoperation of heterogeneous distributed components, it is necessary to construct glue and wrapper code to interconnect the components. Because a project objective is to achieve high quality systems, a goal is to automatically generate the glue/ wrapper code. In order to achieve this, there should be formal rules for interconnecting

components from a specific application domain as well as integration of multiple technology domains, that is, component models. UniFrame uses the Two-Level Grammar (TLG, also called W-grammar) formal specification language (Bryant & Lee, 2002) to specify both types of rules. The TLG formalism is used to specify the components deployed under UniFrame and also the generative rules needed for system assembly. The output of the TLG will provide the desired target code (for example, glue and wrappers for components and necessary infrastructure for the distributed runtime architecture). The UMM formalization establishes the context for which the generative rules may be applied. Bryant, Auguston, Raje, Burt, and Olson (2002) provide further details about the glue/wrapper code generation rules, including a discussion of how the Quality of Service validation code is inserted into the glue code. The general principle is that for each QoS parameter to be dynamically verified, the glue code is instrumented according to the event grammar rules described earlier.

Future Trends

The concept of Business Reference Models "is meant to provide the foundation for common understanding of business processes across the Federal government in a service-oriented manner," enabling an agency to define an enterprise architecture as mandated by law, (Enterprise Architecture SIG, 2003). A significant sector of industry is involved in establishing standards and guidelines on how to enable successful enterprise architecture. The component-based architecture of UniFrame's knowledgebase closely follows these guidelines, incorporating the concepts of Object Management Group's (2002) Model-Driven Architecture as an integral part. Consequently, UniFrame is working toward the realization of an operational framework for enterprise architecture and is a source of feedback into the activities necessary.

Many existing component models provide the necessary mechanisms for describing the functional aspects of components but not for the QoS aspects. Standards organizations have recently started to address this weakness. For example, in the fall of 2000, the OMG began issuing a number of Requests for Proposals for UML profiles for modeling QoS in several contexts. UniFrame is addressing some of these QoS issues and is making efforts (via presentations to different OMG task forces) to ensure that its research is aligned with emerging industry standards.

The creation of the Business Line and Service-Oriented knowledgebase will largely continue to be a human endeavor aided by CASE tools because humans determine what constitutes the problems they must solve. However, the System Generation Process could be accomplished mostly automatically for any problem in a given knowledgebase. The person who formulates the requirements for the DCS will need to do so in the knowledgebase's terminology. The degree to which this can be made to match the typical user's terminology remains a research area.

Huang (2003) implemented a prototype of the UniFrame System Generation Process with the UniFrame Resource Discovery Service. Because of the labor involved in constructing the knowledgebase, it was limited to a small banking case study. Experimental studies

proved efficient, user communication issues were easily managed, and QoS values were calculated. The automated creation of bridges and glue/wrapper code and using FORMAN to insert the code into them for the QoS computations remain to be incorporated into the implementation.

Conclusion

This chapter has described the UniFrame process for constructing distributed computing systems and has shown how it facilitates achieving the current goals of government and industry in rapidly creating high quality computing systems. UniFrame provides a framework within which a diverse array of technologies can be brought to achieve these ends. These include software engineering practices, such as rapid, iterative, and incremental development. Its business line, service-oriented, model-driven architecture based on components is a realization of the movement to provide mutability, quick development, and conservation of resources. A knowledgebase of component-based, predefined and tested designs for distributed computing systems, event traces for empirical testing, and quality of service prediction and calculation are tools it utilizes for increasing quality assurance. UniFrame decouples the requirements analysis and system assembly activities from the problem of collecting appropriate components published on the network. Its novel resource discovery service facilitates the efficient acquisition of components meeting stated specifications. It provides a mechanism for seamlessly bridging components of different models, such as RMI and CORBA, to support the construction of heterogeneous, distributed computing systems having platform-independent definitions. The UniFrame project is also investigating techniques and patterns related to using quality of service parameters during the design of components and integrated systems to create high assurance distributed computing systems.

Acknowledgments

References

Auguston, M. (1995). *Program behavior model based on event grammar and its application for debugging automaton.* In M. Ducassé (Ed.), Proceedings of the 2nd International Workshop on Automated and Algorithmic Debugging (AADEBUG'95) (pp. 277-291), Rennes: Université de Rennes.

Batory, D., & Geraci, B. (1997). Component validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering, 23*(2), 67-82.

Beugnard, A., Jezequel, J., Plouzeau, N., & Watkins, D. (1999). Making components contract aware. *IEEE Computer*, *32*(7), 38-45.

Brahnmath, G. (2002). *The UniFrame Quality of Service Framework*. Unpublished master's thesis, Indiana University Purdue University, Indianapolis, IN, United States. Retrieved August 8, 2004: *http://www.cs.iupui.edu/uniFrame/*

Brown, A. (1999). Building systems from pieces with component-based software engineering. In P. Clements (Ed.), *Constructing superior software* (Chapter 6). Indianapolis, IN: MacMillan Technical.

Bryant, B. R., Auguston, M., Raje, R. R., Burt, C. C., & Olson, A. M. (2002). *Formal specification of generative component assembly using two-level grammar.* Proceedings of SEKE 2002, 14th International Conference on Software Engineering and Knowledge Engineering (pp. 209-212). Los Alamitos: IEEE Press.

Bryant, B. R., & Lee, B.-S. (2002). *Two-Level grammar as an object-oriented requirements specification language.* Proceedings of HICSS-35, the 35th Hawaii International Conference on System Sciences (p. 280). Los Alamitos, CA: IEEE Press. Retrieved August 8, 2004: *http://www.hicss.hawaii.edu/HICSS_35/HICSSpapers/PDFdocuments/STDSL01.pdf*

Crnkovic, I., Schmidt, H., Stafford, J., & Wallnau, K. (Eds.). (2003). Proceedings of the 6th Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction. 25th International Conference on Software Engineering (ICSE). Los Alamitos, CA: IEEE Press. Retrieved August 8, 2004: *http://www.csse.monash.edu.au/~hws/cgi-bin/CBSE6*

Enterprise Architecture SIG, Industrial Advisor Council (IAC). (2003a, March). Business line architecture and integration. Retrieved August 8, 2004: *http://216.219.201.97/documents_presentations/index.htm*

Enterprise Architecture SIG, Industrial Advisor Council. (2003b, March). (IAC). Succeeding with component-based architecture in e-government. Retrieved August 8, 2004: *http://216.219.201.97/documents_presentations/index.htm*

Fritzson, P., Auguston, M., & Shahmehri, N. (1994). Using assertions in declarative and operational models for automated debugging. *The Journal of Systems and Software, 25*, 223-239.

Griss, M. L. (2001). Product line architectures. In G. T. Heineman, & W. T. Councill (Eds.), *Component-based software engineering: Putting the pieces together* (pp. 405-420). Boston: Addison-Wesley.

Heineman, G. T., & Councill, W. T. (Eds.). (2001). *Component-based software engineering: Putting the pieces together*. Boston: Addison-Wesley.

Huang, Z. (2003). *The UniFrame system-level generative programming framework.* Unpublished master's thesis, Indiana University Purdue University, Indianapolis, IN, United States. Retrieved August 8, 2004: *http://www.cs.iupui.edu/uniFrame*

Java Remote Method Invocation – Distributed computing for Java. (2003, October 2). Retrieved August 8, 2004: *http://java.sun.com/marketing/collateral/javarmi.html*

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, *21*(7), 558-565.

Lumpe, M., Schneider, J., Nierstrasz, O., & Achermann, F. (1997). *Towards a formal composition language.* In G. T. Leavens & M. Sitamaran (Eds.), Proceedings of the 1st ESEC Workshop on Foundations of Component-Based Systems (pp. 178-187). Heidelberg: Springer-Verlag.

Microsoft .Net Framework: Technology overview. (2003, October 2). Retrieved August 8, 2004: *http://msdn.microsoft.com/netframework/technologyinfo/overview/*

Object Management Group. Model-Driven Architecture™, the architecture of choice for a changing world. (2002, March 12). Retrieved August 8, 2004: *http://www.omg.org/mda*

Raje, R. (2000). *UMM: Unified Meta-object Model for open distributed systems.* Proceedings of the Fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP 2000) (pp. 454-465). Los Alamitos, CA: IEEE Press.

Raje, R., Auguston, M., Bryant, B., Olson, A., & Burt, C. (2001). *A unified approach for integration of distributed heterogeneous software components.* Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration, SEAC technical report (pp. 109-119). Monterey, CA: U.S. Naval Postgraduate School. Retrieved August 8, 2004: *http://www.cs.iupui.edu/uniFrame/*

Raje, R., Bryant, B., Olson, A., Auguston, M., & Burt, C. (2002). A quality-of-service-based framework for creating distributed heterogeneous software components. *Concurrency and Computation: Practice and Experience, 14,* 1009-1034.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). *The Unified Modeling Language reference manual*. Reading, MA: Addison Wesley.

Schmidt, D. (2003, October 2). Overview of CORBA. Retrieved August 8, 2004: *http://www.cs.wustl.edu/~schmidt/corba-overview.html*

Siram, N., Raje, R., Olson, A., Bryant, B., Burt, C., & Auguston, M. (2002). *An architecture for the UniFrame Resource Discovery Service.* Proceedings of the 3rd International Workshop of Software Engineering and Middleware: Vol. 2596. Lecture Notes in Computer Science (pp. 20-35). Heidelberg: Springer-Verlag.

Sun, C. (2003). *QoS composition and decomposition models in UniFrame.* Unpublished master's thesis, Indiana University Purdue University, Indianapolis, IN, United States. Retrieved August 8, 2004: *www.cs.iupui.edu/uniFrame*

Szyperski, C., Gruntz, D., & Murer, S. (2002). *Component software - Beyond object-oriented programming*. (2nd ed.). Boston: Addison-Wesley/ACM Press.

Warmer, J., & Kleppe, A. (2003). *The Object Constraint Language*. (2nd ed.). Boston: Addison-Wesley.

Weck, W. (1997, June). Independently extensible component frameworks. In M. Mühlhäuser (Ed.), *Proceedings of the 1st International Workshop on Component-*

Oriented Programming (European Conference on Object-Oriented Program-ming, Jyväskylä, Finland), *Special Issues in Object-Oriented Programming* (pp. 177-188). Heidelberg: Springer-Verlag.